Schedae Informaticae Vol. 24 (2015): 197–210
doi: 10.4467/20838476SI.16.017.4358

Visualizing CPU Microarchitecture

Tomasz Wojtowicz
Department of Computer Sciences and Computer Methods, Pedagogical University
ul. Podchorążych 2, 30-084 Kraków, Poland
e-mail: *tomaswoj@gmail.com*

Abstract. Deep understanding of microprocessor architecture, its internal structure and mechanics of its work is essential for engineers in the fields like computer science, integrated circuit design or embedded systems (including microcontrollers). Usually the CPU architecture is presented at the level of ISA, functional decomposition of the chip and data flows. In this paper we propose more tangible, interactive and effective approach to present the CPU microarchitecture. Based on the recent advancements in simulation of MOS6502, one of the most successful microprocessor of all times, that started the personal computing revolution, we present the CPU visualisation framework. The framework supports showing CPU internals at various levels (from single transistor, through logic gates, ending with registers, operation decoders and ALU). It allows for execution of real code and detailed analysis of fetch–decode–execute cycle, measurement of cycles per operation or measurement of the CPU activity factor. The analysis means provided by this framework will also enable us to propose the transistor level simulation speed improvements to the model in the future.

Keywords: CPU, microarchitecture, simulation, registers, pipeline, activity, 6502.

1. Introduction

There are two interesting trends emerging recently in the computing. One of those trends is related to the fact, that computer science industry achieved a certain level of maturity and acknowledges that over the decades it created a vast pool of works (software and applications in general). It is becoming a common understanding, that

all this software represents a certain value (technical and cultural) and there are some efforts started to preserve it for future generations [1,2]. This means there is a need of being able to run that software in the environment that hardware wise and operating system wise is very far from the original platform the software was written for. While over the years many ISA level emulators have been created – engineers are now in pursue of 'high fidelity' emulation platform [3, 4].

One of the means to achieve such high accuracy of emulation is to build a low-level simulator for often forgotten and still proprietary legacy platforms. To reach that goal the legacy platforms are analysed at the very low level, for example by decapping integrated circuits [5–7] and based on their physical layout creating models that mimic almost exactly the way those machines worked.

Another trend is a rise of ubiquitous computing [8], that involves wearable computers, low-power, low-complexity embedded sensors and microcontrollers. The characteristic of those devices is that they are far less complex than the desktop PCs we are using, and to get the most of it – code needs to be optimised to the great levels. From the programmers perspective this resembles the rise of the microcomputers in the late 1980s [9–11] and in this sense technically links directly to the above trend on software preservation.

Both of the trends raise the need for better understanding of how the certain integrated circuit, either a legacy CPU or a dedicated chip or some small sensor, works internally. Knowing in detail the microarchitecture enables the programmers to write more efficient software or more accurate simulator for a given platform.

An additional driver for effective visualisation of the chip at the low levels is that simulation techniques running at transistor or gate-level, while accurate, are relatively slow comparing to ISA level emulation. An insight of how integrated circuit works internally, augmented with some tools measuring the statistics of the simulation model performance, will be an aid in improving the simulation speed.

In this paper, we propose an enhanced CPU visualisation framework, on top of the simulator running at the transistor level. We are presenting a framework architecture, describing how its functionality differs from the previous approaches. We will show in detail certain aspects of the visualisation that may aid the engineers in both reverse engineering the legacy software or writing new software. We will highlight some metrics that can be gathered on top of existing simulation models to speed it up in the future.

2. Previous works

The presentation of CPU microarchitecture have been studied since the beginnings of computing. In the early ages the amount of computing elements (relays, switches and later on transistors) was relatively low so it could often fit to the smallest details on a single blueprint. Since the introduction of transistors their volume grew rapidly over time following the Moore law. To be able to show it effectively abstraction levels are introduced. A CPU is cut into functional components, like ALU, operation decoders,

internal registers, internal buses, etc. This approach is widely present in the literature [12,13]. What is described are usually abstract models, and as such – one cannot really write programs for it nor observe how they work in practise. There are works that focus on some specific aspects – like memory subsystem and cache misses simulation [14, 15]. On the other hand there are efforts in the literature that build computers bottom up, from the level of single transistors and logic gates [16] but again, these computers are just theoretical. Full, real CPU simulators with visualisation, that can run real software are emerging but still rare [5, 7].

3. Proposed approach

In this paper, we propose an extension to existing approaches, specifically working on top the simulator developed in [17]. By introducing accelerated, OpenGL based visualisation, combined with additional semantics being provided to the framework (functional components and overlays) we are significantly increasing the visibility of the microarchitecture of studied CPU. Moreover, a statistics generation module have been introduced, that works on top of the simulation engine, gathers key performance characteristics during simulation and visualises the results in user friendly way.

3.1. Framework architecture

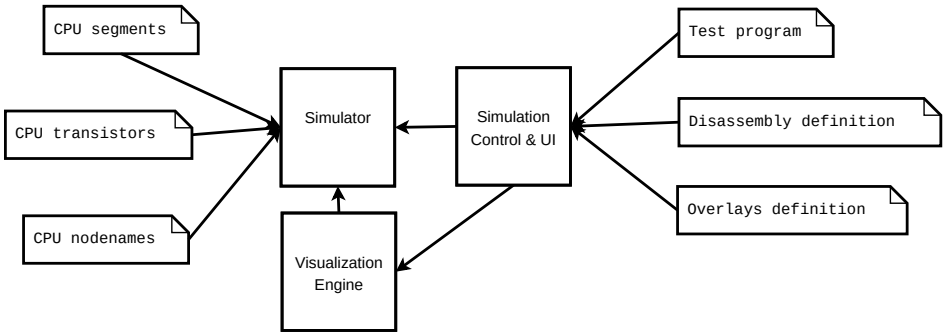


Figure 1. High level of the framework architecture. Key components shown.

The framework consist of several key components. Simulation core is based on Visual6502 simulation implementation [7]. However, for performance reasons but also for higher debugability and flexibility for extension (e.g. swapping the original core with alternatives when testing performance improvements) it was ported from JavaScript to Java. This core at startup is provided with basic netlist type of in-

formation (through the input files) that contain segments (nodes), transistors and named segments (for semantic analysis). It is worth to note, that this information is at very basic level and consists only of interconnection information about these elements. Another key component is the visualisation module. Original works rendered the CPU in 2D only and as being non-accelerated slow down the execution quite significantly. In our implementation we went for OpenGL accelerated view (using Java bindings via JOGL). As a result the main CPU where simulation runs is tied mostly to the actual simulation, while the visualisation bit is handled by the dedicated GPU unit. Moreover OpenGL allows us to easily introduce all sorts of overlays, customised blending, views from various perspectives.

Spanning both the core and the visualisation modules is the simulation control and UI component. It is instrumented by overlay information (with functional components of the CPU, like specific registers, internal buses, CPU pads – used during visualisation), test binary program and disassembly map (of binary code into the assembly language commands).

3.2. Higher level overlays

Typical visualisation of the whole CPUs so far was using mostly physical aspects of the design. CPU was rendered by layers, with different types of colours corresponding to different type of material. There was little indication of the current activity inside the processor.

For more effective microarchitecture analysis we have introduced named bounding boxes, that are corresponding to the functional areas of the CPU. Each bounding box approximates an area (of segments and transistors) responsible for some specific operation in the processor. In our 6502 case we have focused on the CPU pads, internal registers, status registers, ALU, operation decoders, etc. Such a selection allows us to focus on the aspects like operation execution flow (fetch–decode–execute), cycle cost per operation, while discarding more auxiliary aspects like for example interrupt handling.

While the simulator itself is loaded with the chip definition containing only basic connectivity between segments and transistors – the framework can be also provided with component overlays. Definition of those requires some semantic information on the chip, but gives an additional value to the engineer, showing where particular functional units are on the floor chip, and how the data flows through them as the program is executed. Most of functional units on the CPU view are augmented with their current value in the cycle. This is equivalent to the CPU and memory status view (log), but in a more visual form. As there are multiple components inside the CPU, we narrowed the overlay only to selected ones.

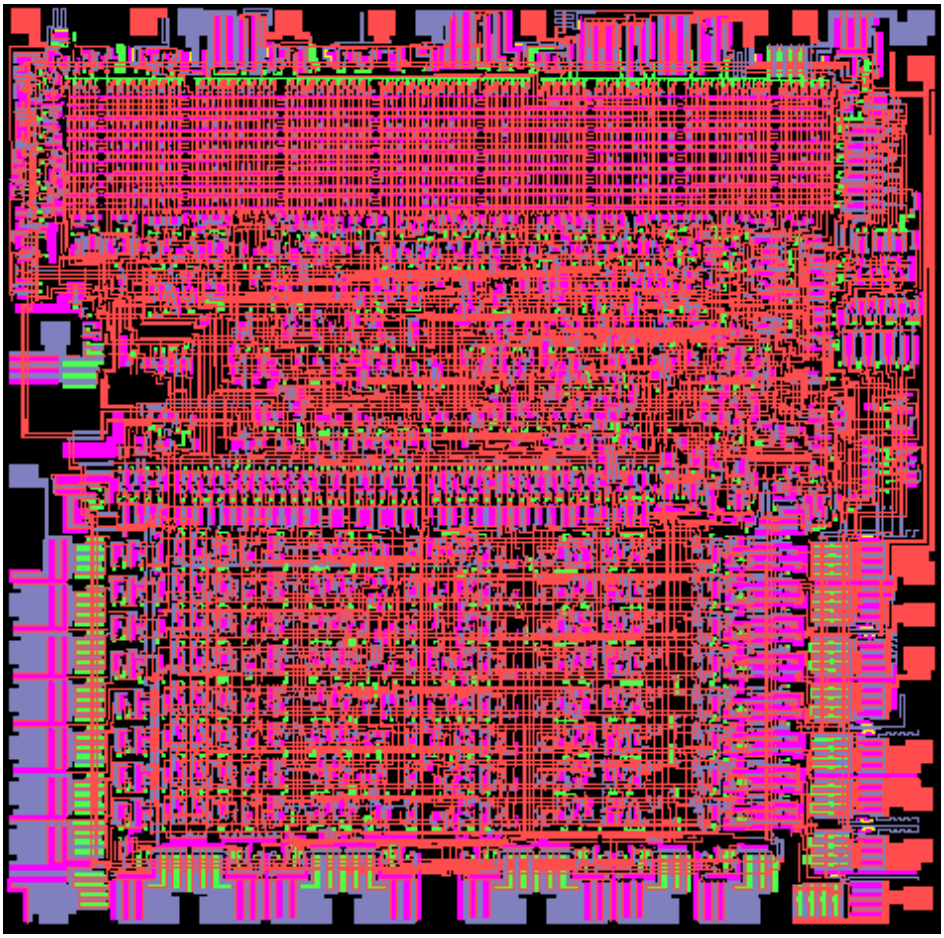


Figure 2. Typical visualisation of the real CPU. Various colours denote various types of layers, like metal, polysilicon, powered diffusion, grounded diffusion.

3.3. Internal datapaths and buses

Another enhancement on the CPU floor plan view is to highlight the internal buses that are part of the processor. Through these buses the data flows between the functional components highlighted in the previous section. This time, to emphasise the directional and transport aspects of these buses they are visualised by accenting the segments they are build of. As these buses are multiple bits wide (in case of 6502 the data bus is 8-bits while the address bus is 16-bits), only a subsets of segments are used. Again – to have a better view on data flows inside the CPU, data paths and buses are presented with their current values in the cycle.

In practise what works best is a combination of functional bounding boxes and data paths between these functional areas. Therefore in our visualisation framework this is all configurable by the user.

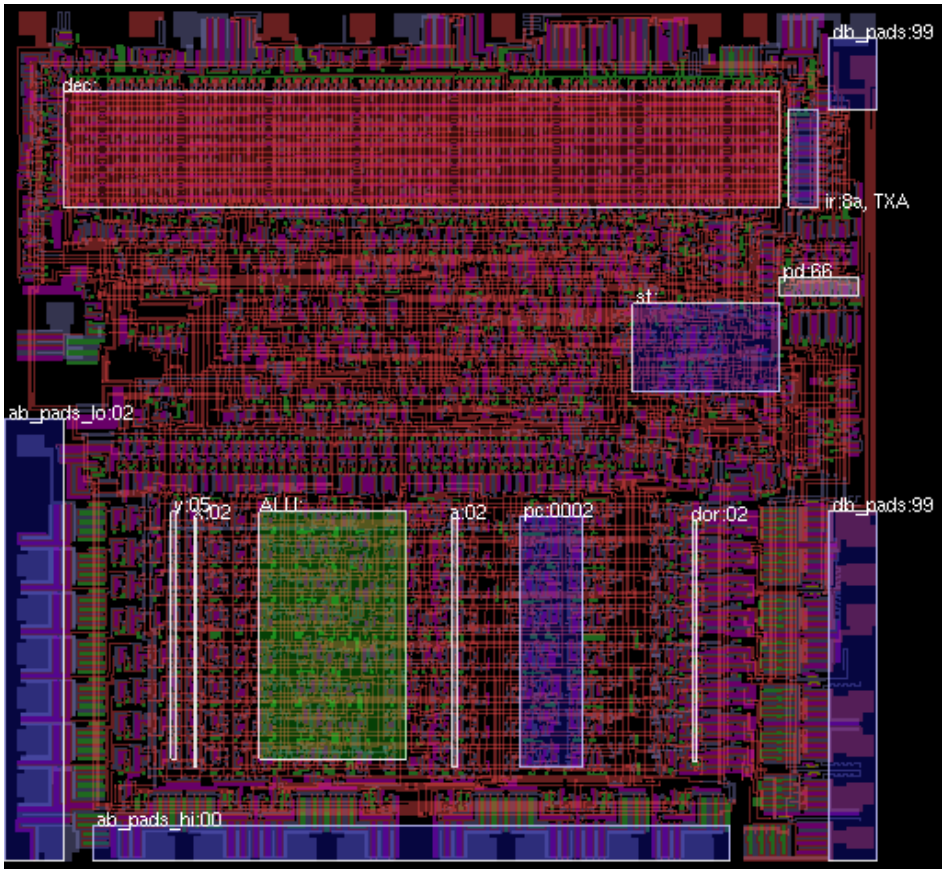


Figure 3. Proposed visualisation approach with functional components layout on the processor floor plan.

3.4. Program and memory view

To be able to trace, how the data goes through the CPU, when one operation concludes and the other starts, it is helpful to see the CPU state in the sequence, cycle after cycle, all in one view. Therefore next to the graphical view of the CPU floor plan with all the important functional areas, data paths and buses we are also providing a more typical view. Each half-cycle we are logging the state of all the above components and provide the outline in a tabular form to the user. We have added an additional colouring here, to give at a glance view, where is the barrier between commands and how the individual commands are structured (especially the ones that fetch or store data from the memory at address provided in the operands). With this colouring one can easily spot, how the CPU passes internally the address vectors and the actual data.

Such view allows to spot early forms of pipelining in the 6502 CPU. Note in Table 3

Table 1. Selection of functional components for the overlay.

Label	Function	Description
IR	Instruction register	Holds the opcode of the current instruction being or just to be executed. It is a perfect place to perform a disassembly to help with the analysis.
Dec	Instruction decoder	A large block of transistors on the north of the chip responsible for decoding the actual operation (held in IR) and instrumenting further the execution pipeline. It is a 130x21 PLA-like ROM. IR area delivers to the PLA the current operation and the current cycle within the operation.
St	Status register	One of the key register in any CPU. Contains multiple flags that indicate overflows from recent operations, comparison results for branching, etc.
PC	Program counter	Internal register that points the program execution to the specific place in memory.
db_pads	External data bus pads	Connecting the CPU to the memory subsystem. Used for reading data from memory or writing data to memory.
ab_pads	External address bus pads	Points the CPU to the specific memory address. Used for memory read/write with values going through the data bus pads.
a, x, y	Internal CPU registers	General purpose registers used in various operations. X and Y allow for multiple offset addressing modes.
ALU	ALU area of the chip	Segments belonging to Arithmetic Logic Unit. Implementing operations like additions, increments, etc.
ab_lo, ab_hi	Internal address buses	Used to pass the address from registers to the external address pads.
sb, idb	Internal datapath and bus	Used to pass the data between registers, and from datapads to the registers.
irq	Interrupt handling	Several segments of the IRQ handling component inside the CPU. Picked for illustrative purposes, to show where they reside on the CPU floor plan.
ops_PLA	Operations decoding	Segments responsible for decoding of couple of operations. Picked for illustrative purposes to show how the operation logic cuts through the PLA region.

that in 1st cycle data bus already contains the next command to be executed (LDA), while the internal address bus is already provided with the address $0001 to fetch the older byte of the LDA operand (0xFE). In the very next cycle (2) the data bus

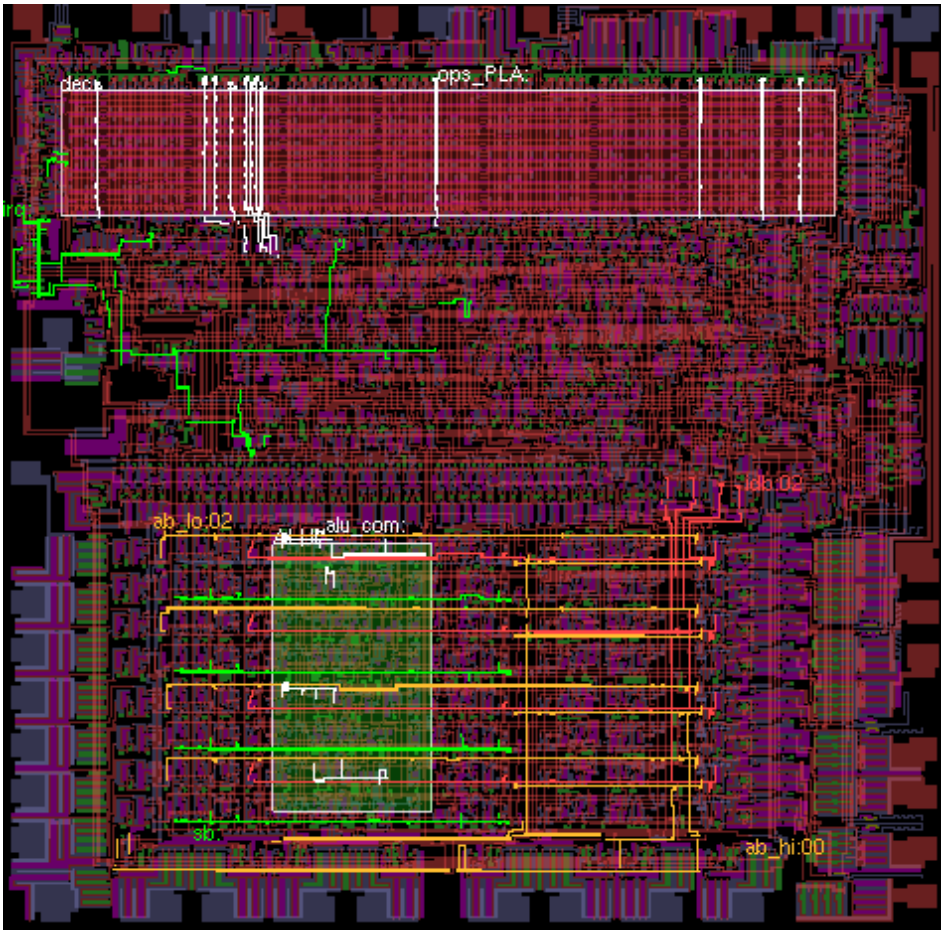


Figure 4. Proposed visualisation approach with internal buses and datapaths shown, combined with overlay for selected functional components.

already fetches the operand while IR is just set to LDA. At the same time the internal address bus is set to fetch the younger byte of the LDA operand (from address 02). This address is exposed on AB in cycle 3 and fetching of the actual data (0x00) is in cycle 4. Such detailed analysis can be also augmented with visual indicators, that highlight which parts of the chip are active at which stages, what will be shown in 3.6.

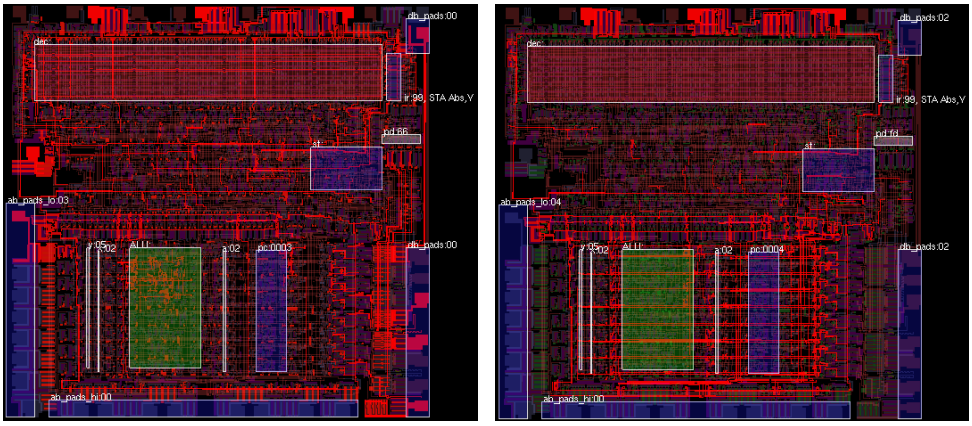
3.5. Activity visualisation

One of the novel features we introduced to the visualisation is showing the segment activity over clock cycle. We are tracking which segments are changing state, and

Table 2. Short 6502 program excerpt (including PC, binary and assembly source code).

Program Counter	Binary	Assembler
0x0000	AD	LDA $00FE
0x0001	FE	
0x0002	00	
0x0003	8D	STA $0000
0x0004	00	
0x0005	00	
0x0006	AD	LDA $00FE

highlight them on the chip layout. When combined with functional component layouts (like ALU, or instruction decoder (PLA)) it can quite well show the engineer – what parts of the CPU are involved in which exact activities in each cycle.



(a) **(b)**

Figure 5. Internal CPU activity, combined with functional component overlays. Red segments denote segments that changed state comparing to the recent cycle in two consecutive cycles (a) and (b). Note that on the left most activity is in the ALU area and external data pads, while on the right most activity is related to internal CPU data transfers.

This, combined with the continuous CPU state information available to the user, can quite effectively explain why certain operations take just 2 cycles (4 half-cycles) and why some other operations require up to 6 cycles (12 half-cycles).

Table 3. Internal CPU states during execution of LDA $00FE command followed by STA $0000 command. Part of the program from 2. Note how the STA command is interleaved with the 2nd half of the LDA command. Such colouring highlights the data flow within the command execution (fetch command, fetch operands, fetch data – if necessary, perform the command, fetch new command).

Timing		Key registers						Internal buses			Pads	
Cycle	Clk	PC	A	X	Y	IR	SP	IDB	AB	lAB	AB	DB
1	(+)	0000	AA	00	00	00:BRK	FD	FF	00 01	00 00	0000	AD
2	(−)	0001	AA	00	00	AD:LDA ABS	FD	FF	00 01	00 01	0001	FE
3	(+)	0001	AA	00	00	AD:LDA ABS	FD	FF	00 02	00 01	0001	FE
4	(−)	0002	AA	00	00	AD:LDA ABS	FD	FE	00 02	00 02	0002	00
5	(+)	0002	AA	00	00	AD:LDA ABS	FD	FF	FF FE	00 02	0002	00
6	(−)	0003	AA	00	00	AD:LDA ABS	FD	FF	00 FE	00 FE	00FE	00
7	(+)	0003	AA	00	00	AD:LDA ABS	FD	FF	00 03	00 FE	00FE	00
8	(−)	0003	00	00	00	AD:LDA ABS	FD	00	00 03	00 03	0003	8D
9	(+)	0003	00	00	00	AD:LDA ABS	FD	00	00 04	00 03	0003	8D
10	(−)	0004	00	00	00	8D:STA ABS	FD	00	00 04	00 04	0004	00
11	(+)	0004	00	00	00	8D:STA ABS	FD	FF	00 05	00 04	0004	00
12	(−)	0005	00	00	00	8D:STA ABS	FD	00	00 05	00 05	0005	00
13	(+)	0005	00	00	00	8D:STA ABS	FD	FF	FF 00	00 05	0005	00
14	(−)	0006	00	00	00	8D:STA ABS	FD	00	00 00	00 00	0000	00
15	(+)	0006	00	00	00	8D:STA ABS	FD	FF	00 06	00 00	0000	00
16	(−)	0006	00	00	00	8D:STA ABS	FD	00	00 06	00 06	0006	AD
17	(+)	0006	00	00	00	8D:STA ABS	FD	00	00 07	00 06	0006	AD
18	(−)	0007	00	00	00	AD:LDA ABS	FD	00	00 07	00 07	0007	FE

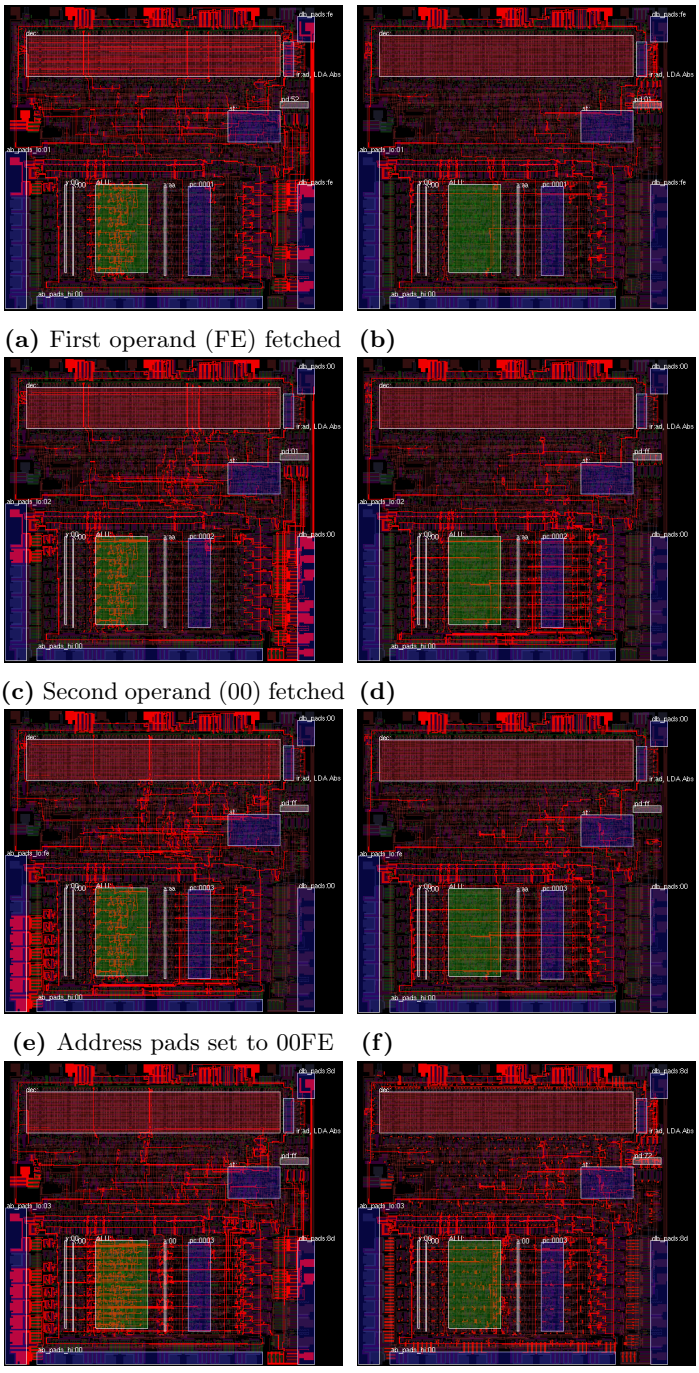
3.6. Using activity overlays to show the pipelining

Below is a CPU activity during the execution of LDA $00FE command and subsequent loading of STA $0000 command (based on the example from 3.4.).

3.7. Measuring simulator performance

The original 6502 CPU was usually running between 1 and 1.5 MHz. The initial version of Visual6502 simulator [7] was running at approximately 25 Hz. Thanks to recent years advancements on JavaScript runtime in web browsers, the simulator can now reach around 250 Hz (with no visualisation enabled). A more optimised C port of the simulator can run at approximately 1 kHz on a modern PC.

The fastest software simulation up to date is based on reverse engineered VHDL model of the original chip schematics and can run as fast as 4 kHz (using Verilator software). There are of course FPGA based implementations developed since the



(a) First operand (FE) fetched **(b)**

(c) Second operand (00) fetched **(d)**

(e) Address pads set to 00FE **(f)**

(g) **(h)**
Value from $00FE loaded to A

Figure 6. Activity map of LDA $00FE command (that takes 4 cycles) followed by the STA $0000 command.

publication of [7] that can run at 1 MHz and more (and can be even integrated into working Atari computer – with all the other dedicated chips working just fine with the FPGA implementation of the 6502 CPU). Our model implementation written in Java can run around 500–700 cycles per second with visualisation enabled.

The challenge with transistor level simulator is, while its pretty accurate in between the cycles, it requires multiple iterations to reach a stable status within a single cycle. On real world programs it can be as high as 30 iterations (but also as low as 2-5). The proposed framework supports gathering of such statistics.

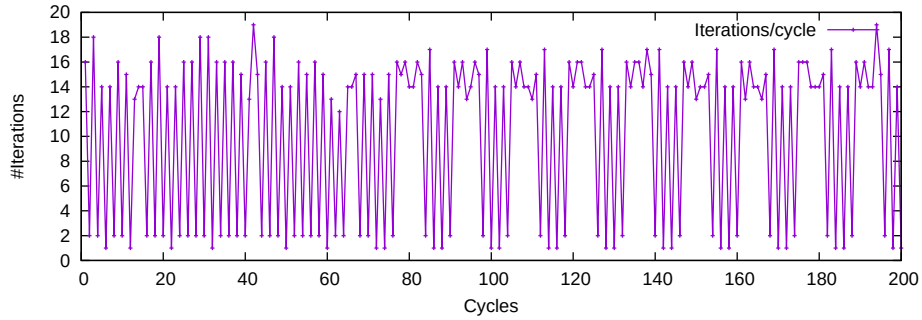


Figure 7. Statistics gathered from the simulator: how many iterations were required to stabilise the CPU state in the cycle.

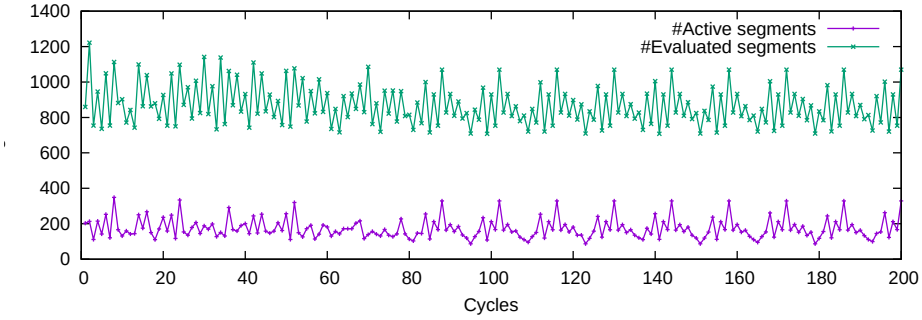


Figure 8. Statistics gathered from the simulator: how many segments had to be evaluated in the cycle against the number of segments that actually changed the state (active). The total amount of segments in 6502 CPU is 1704.

The statistics collected so far indicate, that the activity factor for 6502 is around 10%. This means that on average one tenth of segments change state over each clock cycle. Another metric shows that the current version of the simulator evaluates approximately 5 times more segments, than actually change state over cycle. Taking this and the average amount of iterations needed currently to stabilise the state of the CPU we believe there is a significant potential to speed up the simulation by a factor or two.

4. Further work discussion

The proposed framework addresses quite well the need of straightforward, interactive and dynamic simulation framework for architecture learning purposes. It provides the engineer many capabilities not available so far without a real equipment. It gives an easy and efficient insight on how the microprocessor works.

One of the future work directions is definitely optimising the simulation core itself in terms of performance. A framework is developed in such a way, that it is possible to swap the simulation core with an alternative, test it using some asserts and traces at a single instruction level. Collected measurements indicate there is still a lot of potential for improvements in the way the current model works. Moreover – so far no spatial characteristics were leveraged (fixed topology of the CPU) nor speculative execution was exercised (predict the next state based on the gathered data from previous executions). Another direction of work can be expanding the framework to support any CPU core (like original Motorola 6800, or Intel 4004/8008, or others) – if only the basic netlist information is available.

5. References

[1] Matthews B., Shaon A., Bicarregui J., Jones C., *A framework for software preservation.* The International Journal of Digital Curation, 2010, 5(1), pp. 91–105.

[2] Owens T., *Preserving.exe: Towards a National Strategy for Software Preservation. NDIIPP.* http://www.digitalpreservation.gov/multimedia/documents/PreservingEXE_report_final101813.pdf, 2013 [Accessed 7-July-2014].

[3] Adam, *DICE – Digital Integrated Circuit Emulator.* http://adamulation.blogspot.com/, 2012 [Accessed 5-July-2014].

[4] Byuu, *Accuracy takes power: one mans 3GHz quest to build a perfect SNES emulator.* http://arstechnica.com/gaming/2011/08/accuracy-takes-power-one-mans-3ghz-quest-to-build-a-perfect-snes-emulator/, 2011 [Accessed 5-July-2014].

[5] Aspray W., *The intel 4004 microprocessor: What constituted invention?* IEEE Annals of the History of Computing, 1997, 19(3), pp. 4–15.

[6] Faggin F., *Intel 4004 - 35th Anniversary Project.* http://www.4004.com/, 2005 [Online; accessed 5-July-2014].

[7] James G., Silverman B., Silverman B., Visualizing a classic cpu in action: the 6502. In: *SIGGRAPH Talks*, ACM, 2010.

[8] Zhou Y., Xu T., David B., Chalon R., *Innovative wearable interfaces: an exploratory analysis of paper-based interfaces with camera-glasses device unit.* Personal and Ubiquitous Computing, 2014, 18(4), pp. 835–849.

[9] Bagnall B., *Commodore: A Company on the Edge*, 2010.

[10] Maher J., *The Future Was Here: The Commodore Amiga (Platform Studies).* The MIT Press, Cambridge, 2012.

[11] Vendel C., Goldberg M., *Atari Inc.: Business is Fun.* Syzygy Press, New York 2012.

[12] Hanson D.F., A vhdl conversion tool for logic equations with embedded d latches. In: Proceedings of the 1995 Workshop on Computer Architecture Education. WCAE-1 '95, New York, ACM, 1995.

[13] Tanenbaum A.S., *Structured Computer Organization (5th Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[14] Drepper U., *What every programmer should know about memory.* http://lwn.net/Articles/250967/, 2007 [Accessed 5-July-2014].

[15] Inc. O., *Missing cache visualization.* http://www.overbyte.com.au/misc/Lesson3/CacheFun.html, 2010 [Accessed 5-July-2014].

[16] Stokes J., *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture.* ArsTechnica Library, San Francisco, 2006.

[17] James G., Silverman B., Silverman B., *Visual 6502 CPU simulator.* http://www.visual6502.com/, 2011 [Accessed 5-July-2014].